Open in app ↗

Search

Blob Store

Suresh Podeti

7 min read · Sep 28, 2023

Listen Share More

Photo by CHUTTERSNAP on Unsplash

Introduction

Blob store is a storage solution for unstructured data. We can store photos, audio, videos, binary executable codes, or other multimedia items in a blob store. Every type of data is stored as a **blob**. It follows a flat data organization pattern where **there are no hierarchies**, that is, directories, sub-directories, and so on.

Mostly, it's used by applications with a particular business requirement called **write once, read many (WORM)**, which states that data can only be written once and that no one can change it. Blobs that are **written can't be modified. Instead** of modifying, we can **upload a new version** of a blob if needed.

Requirements

- **Availability:** Our system should be highly available.
- **Durability:** The data, once uploaded, shouldn't be lost unless users explicitly delete that data.
- **Scalability:** The system should be capable of handling billions of blobs.
- **Throughput**: For transferring gigabytes of data, we should ensure a high data throughput.
- **Consistency:** The system should be strongly consistent. Different users should see the same view of a blob.

Design

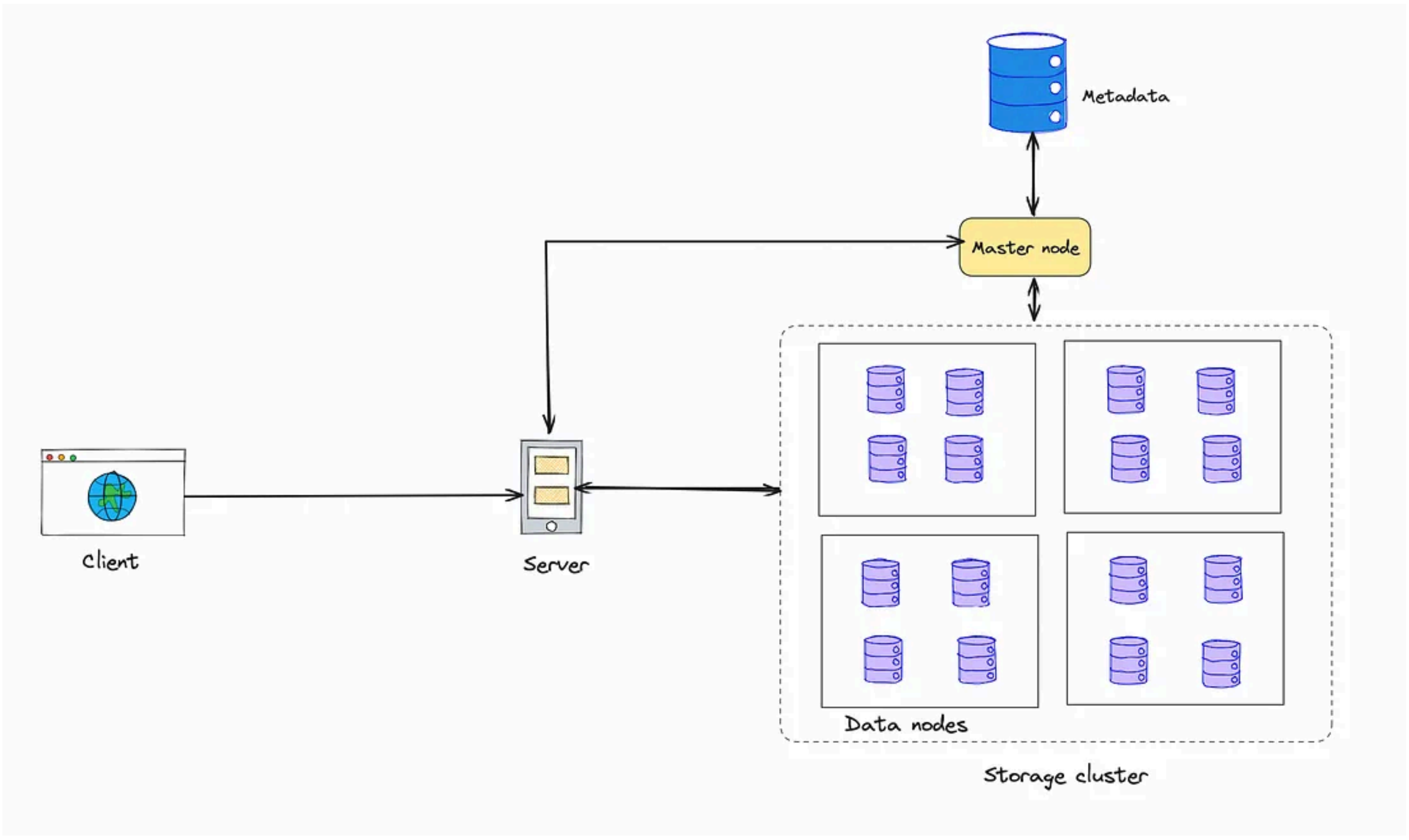

Fig 1.0: Design of blob store

Components

- **Data nodes:** Data nodes hold the actual blob data. It's also possible that they contain a part of the blob's data. Blobs are split into small, fixed-size pieces called **chunks.** A data node can accommodate all of the chunks of a blob or at least some of them.
- **Master node:** A master node is the core component that manages all data nodes. It stores information about storage paths and the access privileges of blobs.
- **Metadata storage:** Metadata storage is a distributed database that's used by the master node to store all the metadata. Metadata consists of account metadata, container metadata, and blob metadata.

Each of the data nodes in the cluster send the master node a **heartbeat** and a chunk report regularly. The presence of a **heartbeat** indicates that the data node is operational. A **chunk report** lists all the chunks on a data node. If a data node fails to send a heartbeat, the master node **considers that node dead** and then processes the user requests on the replica nodes. The master node **maintains a log of pending operations** that should be replayed on the dead data node when it recovers.

Write a blob

1. The client generates the upload blob request. The front-end server then requests the master node for the data nodes it should contact to store the blob.
2. The master node assigns the blob a unique ID using a unique ID generator system. It then splits the large-size blob into smaller, fixed-size chunks and assigns each chunk a data node where that chunk is eventually stored. The master node determines the amount of storage space that's available on the data nodes using a **free-space management system**.
3. After determining the mapping of chunks to data nodes, the servers write the chunks to the assigned data nodes.
4. We replicate each chunk for redundancy purposes. All choices regarding chunk replication are made at the master node. Hence, the master node also allocates the storage and data nodes for storing replicas.
5. The master node stores the blob metadata in the metadata storage.
6. After writing the blob, a fully qualified path of the blob is returned to the client. The path consists of the user ID, container ID where the user has added the blob, the blob ID, and the access level of the blob.

Read a blob

1. When a read request for a blob reaches the front-end server, it asks the master node for that blob's metadata.

2. Master node looks for the chunks for that blob in the metadata and looks at their mappings to data nodes. The master node returns the chunks and their mappings (data nodes) to the client.

3. The client then reads the chunk data from the data nodes.

The metadata information for reading the blob is **cached** at the client machine (server), so that the next time a client wants to read the same blob, we won't have to burden the master node. Additionally, the client's read operation will be faster the next time.

Delete a blob

Deleting a blob Upon receiving a delete blob request, the master node marks that blob as deleted in the metadata, and frees up the space later using a garbage collector.

Design Considerations

Blob metadata

When a user uploads a **blob**, it's **split into small-sized chunks** in order to be able to support the storage of large files that **can't fit in one contiguous location**, in one data node.

The chunks for a single blob are then stored on different data nodes, and billions of blobs that are kept in storage. The master node has to store all the information about the blob's chunks and where they are stored on blob metadata.

To avoid complexity, the **chunk size is fixed** for all the blobs in a blob store. The chunk size depends on the performance requirements of a blob store. We desire a larger chunk size to maintain small metadata at the master node because a large chunk size results in a higher disk latency, which leads to slower performance. Interestingly, disks can have **almost the same latency** for reading and writing a range of data. This is due to the use of contiguous sectors on the disks, and caching on the disk and on the server by its OS.

Partition data

We can **partition** the blobs based on the **complete path of the blob**. The partition key here is the combination of the account ID, container ID, and blob ID. This helps in co-locating the blobs for a single user on the same partition server, which enhances performance.

The **partition mappings** are maintained by the master node, and these mappings are stored in the distributed metadata storage.

Blob indexing

Finding specific blobs in a sea of blobs becomes more difficult and time-consuming with an increase in the number of blobs that are uploaded to the storage.

To populate the blob index, we use multiple tags, like container name, blob name, upload date and time, and some other categories like the image or video blob, and so on while uploading the blob.

Replication

To keep the data strongly consistent, we synchronously replicate data among the nodes that are used to serve the read requests, read after performing the write operation. To achieve availability, we can replicate data to different regions or data centers after performing the write operation. We don't serve the read request from the other data center or regions until we have not replicated data there.

These are the two levels of replication:

- *Synchronous replication* within a storage cluster.
- *Asynchronous replication* across data centers and regions.

Garbage collection

Since the blob chunks are placed at different data nodes, deleting from many different nodes takes time, and holding a client until that's done is not a viable option. Due to real-time latency optimization, we don't actually remove the blob from the blob store against a delete request. Instead, we just mark a blob as "DELETED" in the metadata to make it inaccessible to the user. The blob is removed later after responding to the user's delete request.

Marking the blob as deleted, but not actually deleting it at the moment, causes internal metadata inconsistencies, meaning that things continue to take up storage space that should be free. These metadata inconsistencies have no impact on the user.

We have a service called a **garbage collector** that cleans up metadata inconsistencies later. The deletion of a blob causes the chunks associated with that blob to be freed. However, there could be an appreciable time delay between the time a blob is deleted by a user and the time of the corresponding increase in free space in the blob store. We can bear this appreciable time delay because, in return, we have a real-time fast response benefit for the user's delete blob request.

Streaming

To stream a file, we need to define how many Bytes are allowed to be read at one time. Let's say we read *X* number of Bytes each time. The first time we read the first *X* Bytes starting from the 0th Byte (0 to X−1) and the next time, we read the next *X* Bytes (*X* to 2X−1).

Evaluation

Availability

We keep four replicas for each blob. Having replicas, we can distribute the request load. If one node fails, the other replica node can serve the request. Moreover, our replica placement strategy handles a whole data center failure and can even handle the situation of region unavailability due to natural disasters.

For write requests, we replicate the data within the cluster in a fault-tolerant way and quickly respond to the user, making the system available for write requests.

To keep the master node available, we keep a **backup of its state**. In the case of a master-node failure, we start a new instance of the master node, initializing it from the saved state.

Durability

The data, once uploaded, is synchronously replicated within a storage cluster. If data loss occurs at one node, we can recover the data from the other nodes.

Scalability

The partitioning and splitting of blobs into small-sized chunks helps us scale for billions of blob requests. Blobs are partitioned into separate ranges and served by different partition servers. The partition mappings specify which partition server will serve which particular blob range requests.

Our system is horizontally scalable for storage. As need for storage arises, we add more data nodes.

Throughput

We save chunks of a blob on different data nodes that distribute the requests for a blob to multiple machines. Parallel fetching of chunks from multiple data nodes helps us achieve high throughput.

Additionally, caching at different layers — the client side, front-end servers, and the master node — improves our throughput and reduces latency.

Consistency

We synchronously replicate the disk data blocks inside a storage cluster upon a write request, making the data strongly consistent inside the storage cluster. This is done on the user's critical path. We serve the subsequent read requests on this data from the same storage cluster until we haven't replicated this data in another data center and or on other storage clusters.

Blobstore Software Architect Software Architecture Distributed Systems

System Design Interview

Edit profile

Written by Suresh Podeti

1.2K Followers

Aspiring Software Architect | SDE III (Golang) @JungleeGames| Ex-Byju's | Co-founder @Rupant Tech. | Ex-Synopsys | I.I.T Kharagpur | 7+ Years of Experience